Designing Maintainability in Software Engineering: a Quantified Approach.

Tom Gilb
Result Planning Limited
Tom.Gilb@INCOSE.org

Abstract.

Software system maintenance costs are a substantial part of the life cycle costs. They can easily steal all available effort away from new development. I believe that this is because maintainability is as good as never systematically engineered into the software. Our so-called software architects bear a primary responsibility for this, but they do not engineer to targets. They just throw in customs and habits that seem appropriate. We need to define our maintainability requirements quantitatively, set investment targets that will pay off, pursue long-term engineered improvement of the systems, and then 'architect' and 'engineer' the resulting system. Other disciplines within systems engineering may already in principle understand this discipline, some may not understand it, some may simply not apply the engineering understanding that is out there

The Maintainability Problem

Software systems are built under high pressure to meet deadlines, and with initial emphasis on performance, reliability, and usability. The software attributes relating to later changes in the software – maintainability attributes are:
• never specified quantitatively up front in the software quality requirements
• never architected to meet the non-specified maintainability quality requirements
• never built to the unspecified architecture to meet the unspecified requirements
• never tested before software release
• never measured during the lifetime of the system.

"A number of people expressed the opinion that code is often not designed for change. Thus, while the code meets its operational specification, for maintenance purposes it is poorly designed and documented " [Dart 93]

In short, there is no engineering approach to software maintainability.

<u>What *do* we do in practice today?</u>
• we might bullet point some high-level objectives ('• Easy to maintain') which are never taken seriously
• we might even decide the technology we will use to reach the vague ideal ("• Easy to maintain through modularization, object orientation and state of the art standard tools")
• larger institutions might have 'software architects' who carry out certain customs, such as

decomposition of the software, choice of software platforms and software tools – generally intended to help – hopefully. But with no specific resulting level or type of maintainability in mind.
• we might recommend more and better tools, but totally fail to suggest an engineering approach [Dart 93].

We could call this a 'craft' approach. It is not engineering or architecture in the normal sense.

Principles of Software Maintainability

I would like to suggest a set of principles about software maintainability, in order to give this paper a framework:

1. **The Conscious Design Principle**: Maintainability must be consciously designed into a system: failure to design to a set of levels of maintainability means the resulting maintainability is both bad and random.

2. **The Many-Splendored Thing Principle**. Maintainability is a wide set of change quality types, under a wide variety of circumstances: so we must clearly define what we are trying to engineer.

3. **The Multi-Level Requirement Principle.** The levels of maintainability we decide to require are partly constraints, a necessary minimum of ability to avoid failure, and partly desirable target levels that are determined by what pays off to invest in.

4. **The Payoff Level Principle.** The levels of maintainability it pays off to invest in, depend on many factors – but certainly on the system lifetime expectancy, the criticality/illegality/cost of not being able to change correctly or change in time, and the cost and availability of necessary skilled professionals to carry out the changes.

5. **The Priority Dynamics Principle.** The maintainability requirements must compete for priority for limited resources with all other requirements. We cannot simply demand arbitrary desired levels of maintainability.

The Engineering Solution

There are many small and less critical software systems where engineering the maintainability would not be interesting, or would not pay off. Nobody cares. This paper is addressed to the vast number of current situations where the total size of software, the growth of software annually, the cost of maintenance annually – are all causing management to wonder – 'Is there a better way?'

The method is straightforward, and well understood engineering in ‘real’ engineering disciplines. In simple terms it is:

1. Define the maintainability requirements quantitatively.
2. Design to meet those requirements, if possible and economic.
3. Implement the designs and test that they meet the required levels.
4. Quality Control that the design continues to meet the required maintainability quality levels, and take action in the case of degradation, to get back to current required levels.

Let us take a simplified tour of the method.

Requirement specification (using ‘Planguage’ [Gilb 2005]:

Bug Fixing Speed:
Type: Software Product Quality Requirement.
Scope: Product Confirmit [Version 12.0 and on]
Ambition Level: Fast enough bug fixing so that it is a non-issue with our customers.
Scale of Measure: Average Continuous Hours from Bug occurs and is observed in any user environment, until it is correctly corrected and sufficiently tested for safe release to the field, and the change is in fact installed at, at least, one real customer, and all consequences of the bug have been recovered from at the customer level.
Meter: QA statistics on bug reports and bug fixes.
Past [Release 10.0] 36 hours <- QA Statistics
Fail [Release 12.0, Bug Level = Major] 6 hours <- QA Directors Plan
Target [Release 12.0, Bug Level = Catastrophic] 2 hours <- QA Directors Plan.
Target [Release 14.0, Bug Level = Catastrophic] 1 hour <- QA Directors Plan.

It should be possible to read this specification, slowly, even for those not trained in Planguage, and to be able to explain exactly what the requirement is.

Notice especially the ‘Scale of Measure’. It encompasses the entire maintenance life cycle from first bug effect observation until customer level correction in practice. That is a great deal more than just some programmer staring at code and seeing the bug and patching it. The corresponding design will have to encompass many processes and technologies.

The Breakdown into Sub-problems

Here is a list of the areas we need to design for, and quite possibly have a secondary target level for each:

1. Problem Recognition Time.
   How can we reduce the time from bug actually occurs until it is recognized and reported?
2. Administrative Delay Time:
   How can we reduce the time from bug reported, until someone begins action on it?

3. Tool Collection Time.
How can we reduce the time delay to collect correct, complete and updated information to analyze the bug: source code, changes, database access, reports, similar reports, test cases, test outputs.

4. Problem Analysis Time.
Etc. for all the following phases defined, and implied, in the Scale scope above.

5. Correction Hypothesis Time

6. Quality Control Time

7. Change Time

8. Local Test Time

9. Field Pilot Test Time

10. Change Distribution Time

11. Customer Installation Time

12. Customer Damage Analysis Time

13. Customer Level Recovery Time

14. Customer QC of Recovery Time

Let us take a look at a possible first draft of some design ideas:
Note: I have intentionally suggested some dramatic architecture, in an effort to meet the radically improve requirement level. The reader need not take any design too seriously. This is an example of trying to solve the problem, using engineering techniques (redundancy) that have a solid scientific history.

1. Problem Recognition Time.
1.1 Automated N-version distinct software comparison [Inacio 1998] at selected critical customer sites, to detect potential bugs automatically.

2. Administrative Delay Time:
2.1 Direct digital report from distinct software discrepancies to our global, 3 zone, 24/7 bug analysis service.

3. Tool Collection Time.
3.1 All necessary tools are electronic, and collection is based on customers installed version and its fixes. The distinct software, bug capture collects local input sequences.

4. Problem Analysis Time.
4.1 The fastest bug analysts are selected based on actual past performance statistics, and rewarded in direct relation to their timing for analyzing root cause, or correct fix.

5. Correction Hypothesis Time

5.1 Same design as 4.1, but applies to correct change specification speed statistics.

6. Quality Control Time

6.1 Rigorous 30 minute or less inspection of change spec by other bug analysts, with reward for finding serious errors according to our standards.

7. Change Time

7.1 Changes are applied in parallel with QC, and modified only if defects found.

8. Local Test Time

8.1 automated based on distinct software (2 independent changes to distinct modules, and running reasonable test sets, until further notice or failure.

9. Field Pilot Test Time

9.1 After 30 minutes successful Local Test, the changes are implemented at a customer pilot site for more realistic testing, in operation, in distinct software safe mode.

10. Change Distribution Time

10.1 All necessary changes are readied and uploaded for customer download, even before Local Tests Begin, and changed only if tests fail.

11. Customer Installation Time

11.1 Customer is given options of manual or automatic changes, under given circumstances

12. Customer Damage Analysis Time

12.1 <local customer solution>. We don't have good automation here. Assume none until proven otherwise. We need to be aware of all reports sent and databases updated that may need correction.

13. Customer-Level-Recovery Time

13. same problem as 12.1. may be highly local and manual. Is it really out of our control?

14. Customer QC of Recovery, Time.

14.1 30-minute QC of recovery results, assisted by our guidelines, and for critical customers our staff, remote or on site.

My main point is that each sub-process of the maintenance operation tends to require a separate and distinct design (1 or more designs each). There is nothing simple like software people seem to believe, that better code structures, coding practices, documentation, and tools will solve the maintenance problem.

Maintainability Concepts

Maintainability in the strict engineering sense is usually taken to mean bug fixing. I have however been using it thus far to describe any software change activity or process. We could perhaps better call it 'software change ability'. Different classes of change, will have different requirements related to them, and consequently different technical solutions. It is important that we be very clear in setting requirements, and doing corresponding design, exactly what types of change we are talking about. The following will give a general set of patterns for defining and distinguishing difference classes of maintenance. But in your real world, you will want to tailor the definitions to your domain. This can most easily be accomplished in the Scale of measure definition, initially. And continued tailoring can be done by defining conditions in the requirement level qualifier. For example:

Code Portability:

Scale: Effort in Hours needed to Port each 1000 Non-Commentary Lines of Code from a defined Home Environment to a defined Target Environment, using defined Tools and defined Personnel.

Goal [Home Environment = {.net, Oracle,} , Target Environment = {Java++, Open Source, Linux} , Tools = Convert Open , Personnel = {Experienced Experts, India}] 60 hours.

154 Competitive Engineering

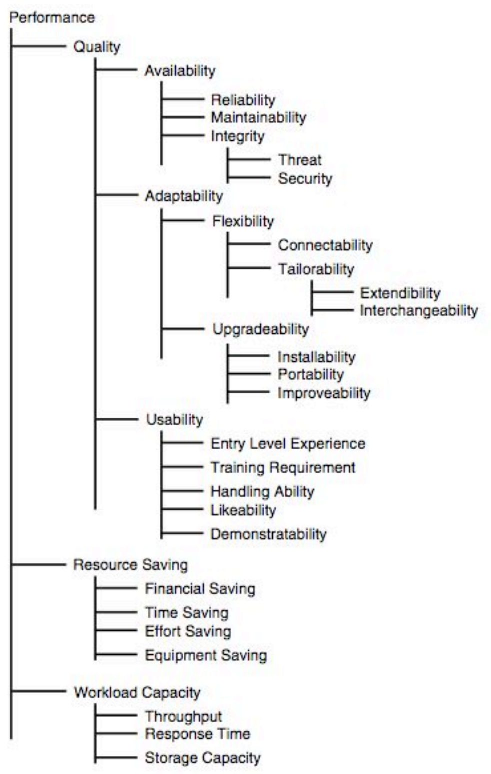


Figure 5.3
One decomposition possibility for performance attributes with emphasis on the detail of the quality attributes.

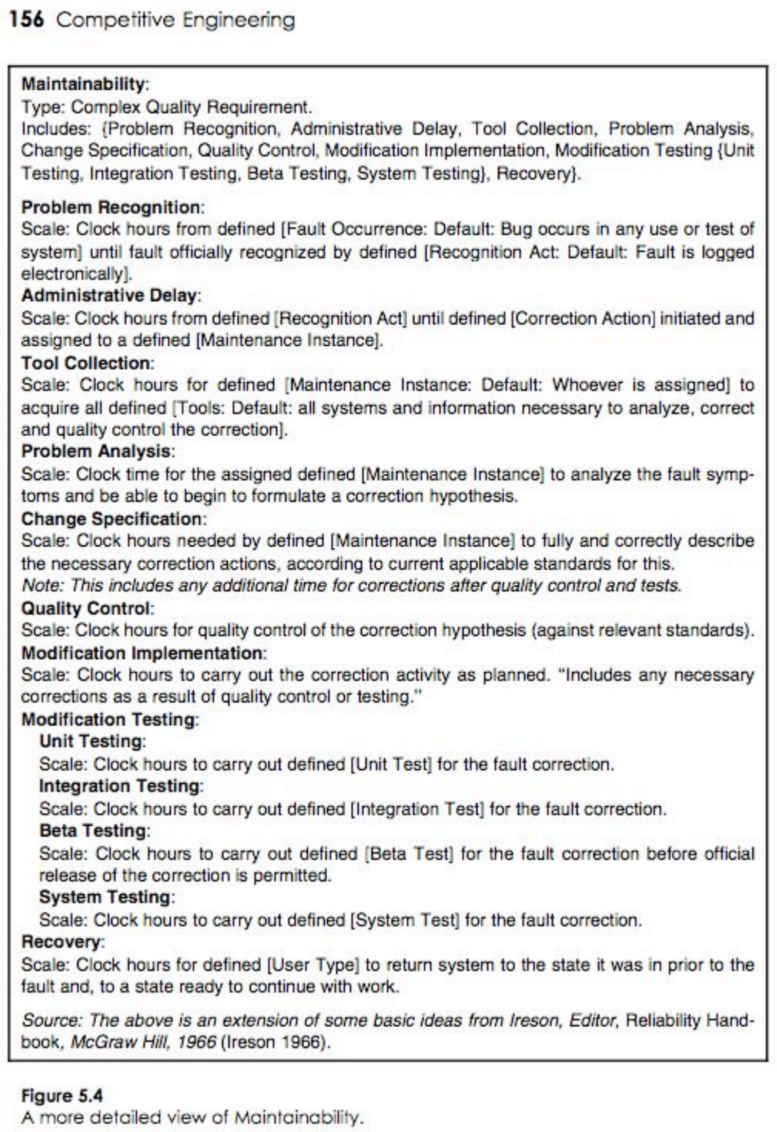
Figure 1: A generic set of performance measures, including several related to change.

The names used are arbitrary choices by the author. They only take on meaning when defined, with a scale of measure. There are no accepted or acceptable standards here, and certainly not for software. Even in hardware engineering, there is an accepted pattern – such as “Scale: Mean Time to Repair”. But it is accepted that we have to define concepts locally, such as the meaning of ‘Repair’. How many of the above 14 sub-processes of maintainability would you include?

Notice that Maintainability in the narrow sense (fix bugs) is quite separate from other 'Adaptability' concepts. This is normal engineering, that places fault repair together with reliability and availability; which all determine the immediate operational characteristics of the system. The other forms of adaptability are more about potential future upgrades to the system, change, rather than repair. They do, both types of change, have in common that our system architecture has to make it easy to change, analyze and test. The system itself is unaware of whether we are changing a fault or improving the system. The consequence is that much of the maintenance design benefits most of the types of maintenance.

Maintainability Measures

Here are some of the general patterns we can use to define and distinguish the different classes of change processes on software. First the 'Bug Fixing pattern (from which we derived the example at the beginning of this paper).



156 Competitive Engineering

Maintainability:
Type: Complex Quality Requirement.
Includes: {Problem Recognition, Administrative Delay, Tool Collection, Problem Analysis, Change Specification, Quality Control, Modification Implementation, Modification Testing {Unit Testing, Integration Testing, Beta Testing, System Testing}, Recovery}.

Problem Recognition:
Scale: Clock hours from defined [Fault Occurrence: Default: Bug occurs in any use or test of system] until fault officially recognized by defined [Recognition Act: Default: Fault is logged electronically].
Administrative Delay:
Scale: Clock hours from defined [Recognition Act] until defined [Correction Action] initiated and assigned to a defined [Maintenance Instance].
Tool Collection:
Scale: Clock hours for defined [Maintenance Instance: Default: Whoever is assigned] to acquire all defined [Tools: Default: all systems and information necessary to analyze, correct and quality control the correction].
Problem Analysis:
Scale: Clock time for the assigned defined [Maintenance Instance] to analyze the fault symptoms and be able to begin to formulate a correction hypothesis.
Change Specification:
Scale: Clock hours needed by defined [Maintenance Instance] to fully and correctly describe the necessary correction actions, according to current applicable standards for this.
Note: This includes any additional time for corrections after quality control and tests.
Quality Control:
Scale: Clock hours for quality control of the correction hypothesis (against relevant standards).
Modification Implementation:
Scale: Clock hours to carry out the correction activity as planned. "Includes any necessary corrections as a result of quality control or testing."
Modification Testing:
Unit Testing:
Scale: Clock hours to carry out defined [Unit Test] for the fault correction.
Integration Testing:
Scale: Clock hours to carry out defined [Integration Test] for the fault correction.
Beta Testing:
Scale: Clock hours to carry out defined [Beta Test] for the fault correction before official release of the correction is permitted.
System Testing:
Scale: Clock hours to carry out defined [System Test] for the fault correction.
Recovery:
Scale: Clock hours for defined [User Type] to return system to the state it was in prior to the fault and, to a state ready to continue with work.

Source: The above is an extension of some basic ideas from Ireson, Editor, Reliability Handbook, *McGraw Hill, 1966* (Ireson 1966).

Figure 5.4
A more detailed view of Maintainability.

Figure 2: Maintainability components, derived from a hardware engineering view, adopted for software.

Here are a generic set of definitions for the 'Adaptability' concepts.

1.2 Adaptability: ‘The efficiency with which a system can be changed.’
Gist: Adaptability is a measure of a system’s ability to change.
Since, if given sufficient resource, a system can be changed in
almost any way, the primary concern is with the amount of
resources (such as time, people, tools and finance) needed to bring
about specific changes (the ‘cost’).

Adaptability: Type: Elementary Quality Requirement.
Scale: Time needed to adapt a defined [System] from a defined
[Initial State] to another defined [Final State] using defined
[Means].
Adaptability: Type: Complex Quality Requirement.
Includes: {Flexibility, Upgradeability}.

1.2.1 **Flexibility**:
Gist: This concerns the ‘in-built’ ability of the system to adapt
or to be adapted by its users to suit conditions (without any
fundamental system modification by system development).
Type: Complex Quality Requirement.
Includes: {Connectability, Tailorability}.

1.2.1.1 **Connectability**: ‘The cost to interconnect the system to
its environment.’
Gist: The support in-built within the system to con-
nect to different interfaces.

1.2.1.2 **Tailorability**: ‘The cost to modify the system to suit its
conditions.’
Type: Complex Quality Requirement.
Includes: {Extendibility, Interchangeability}.

1.2.1.2.1 **Extendibility**:
Scale: The cost to add to a defined [System] a

defined [Extension Class] and defined [Extension
Quantity] using a defined [Extension Means].
‘‘In other words, add such things as a new user or
a new node.’’
Type: Complex Quality Requirement.
Includes: {Node Addability,
Connection Addability,
Application Addability,
Subscriber Addability}.

1.2.1.2.2 **Interchangeability**: ‘The cost to modify use of system components.’
Gist: This is concerned with the ability to modify
the system to switch from using a certain set of
system components to using another set.
For example, this could be a daily occurrence
switching system mode from day to night use.

1.2.2 **Upgradeability**: ‘The cost to modify the system fundamentally; either to install it or change out system components.’
Gist: This concerns the ability of the system to be
modified by the system developers or system support
in planned stages (as opposed to unplanned mainte-
nance or tailoring the system).
Type: Complex Quality Requirement.
Includes: {Installability, Portability, Improveability}.

1.2.2.1 **Installability**: ‘The cost to install in defined conditions.’
This concerns installing the system code and also,
installing it in new locations to extend the system
coverage. Could include conditions such as the instal-
lation being carried out by a customer or, by an IT
professional on-site.

1.2.2.2 **Portability**: ‘The cost to move from location to location.’
Scale: The cost to transport a defined [System] from a
defined [Initial Environment] to a defined [Target
Environment] using defined [Means].
Type: Complex Quality Requirement.
Includes: {Data Portability,
Logic Portability,
Command Portability,
Media Portability}.

1.2.2.3 **Improveability**: ‘The cost to enhance the system.’
Gist: The ability to replace system components with
others, which possesses improved (function, performance, cost and/or design) attributes.
Scale: The cost to add to a defined [System] a defined
[Improvement] using a defined [Means].

Hopefully this set of patterns gives you a departure point for defining those maintenance attributes you might want to control, quantitatively.

The above adaptability definition was use to co-ordinate the work of 5,000 software engineers, and 5,000 hardware engineers, in UK, in bringing out a new product line at a computer manufacturer. It was economically successful for the next 14 years..

The Software Architect Role in Maintainability

The role of the software architect is:

• to participate in clarification of the requirements that will be used as inputs to their architecture process.

• to insist that the requirements are testably clear: that means with defined and agreed scales of measure, and defined required levels of performance.

• to then discover appropriate architecture, capable of delivering those levels of performance, hopefully within resource constraints, and

• estimate the probable impact of the architecture, on the requirements (Impact Estimation)

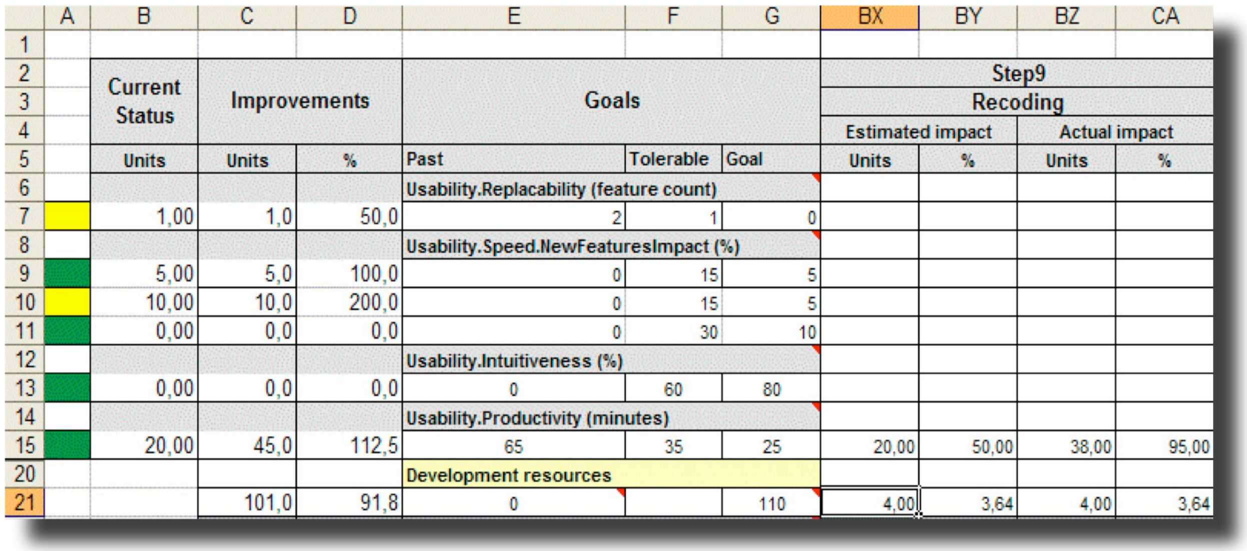
• define the architecture in such detail that the intent cannot be misunderstood by implementers, and the desired effects are bound to be delivered.

• monitor the developing system as the architecture is applied in practice,

• and make necessary adjustments.

• finally monitor the performance characteristics throughout the lifetime of the system, and make necessary adjustments to requirements and to architecture, in order to maintain needed system performance characteristics.

Evaluating Maintainability Designs Using Impact Estimation

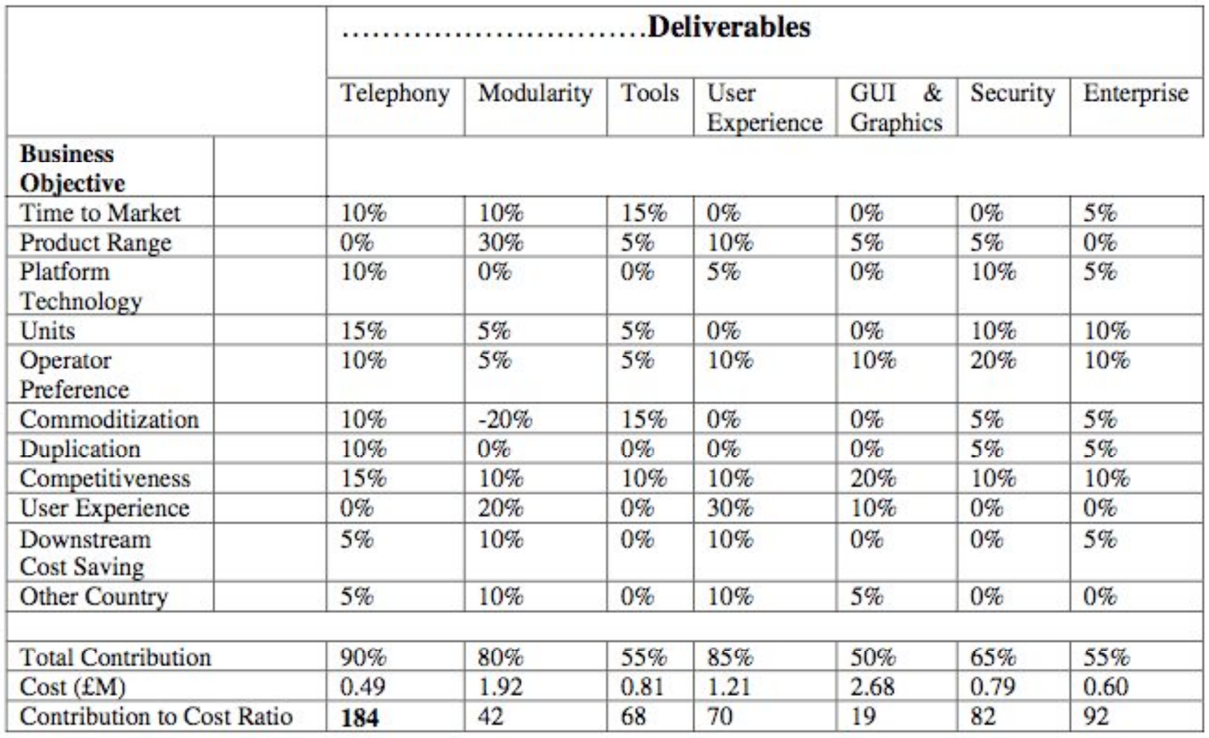


	A	B	C	D	E	F	G	BX	BY	BZ	CA
1											
2		Current Status	Improvements		Goals			Step9			
3								Recoding			
4								Estimated impact		Actual impact	
5		Units	Units	%	Past	Tolerable	Goal	Units	%	Units	%
6					Usability.Replacability (feature count)						
7	🟨	1,00	1,0	50,0	2	1	0				
8					Usability.Speed.NewFeaturesImpact (%)						
9	🟩	5,00	5,0	100,0	0	15	5				
10	🟨	10,00	10,0	200,0	0	15	5				
11	🟩	0,00	0,0	0,0	0	30	10				
12					Usability.Intuitiveness (%)						
13	🟩	0,00	0,0	0,0	0	60	80				
14					Usability.Productivity (minutes)						
15	🟩	20,00	45,0	112,5	65	35	25	20,00	50,00	38,00	95,00
20					Development resources						
21			101,0	91,8	0		110	4,00	3,64	4,00	3,64

Figure 3: a simple Impact Estimation Table spreadsheet used by our client to manage the designs made weekly by a small team of software engineers, for 12 weekly iterations before worldwide software release.

The local design process illustrated above applies to the local, small team, engineering of any performance characteristic, including all maintainability characteristics (see Green Week example below).

The small team (3-5 people) of software engineers (who are responsible to an architect, who is in the weekly loops with them) picks one unfulfilled goal (Improvement % column, less than 100% of Goal level achieved to date) to work on together that week. They brainstorm an number of designs to fill the gap towards 100%, and estimate (Estimated Impact Units, and % to goal) the impact of proposed designs. They then pick the most powerful one or more that can be built in that week, and do it. Independent customer pilot site measurement (Actual Impact) tells them by end of week, how well the design actually worked. They continue that process until that software ships at the end of a Quarter of a Year.

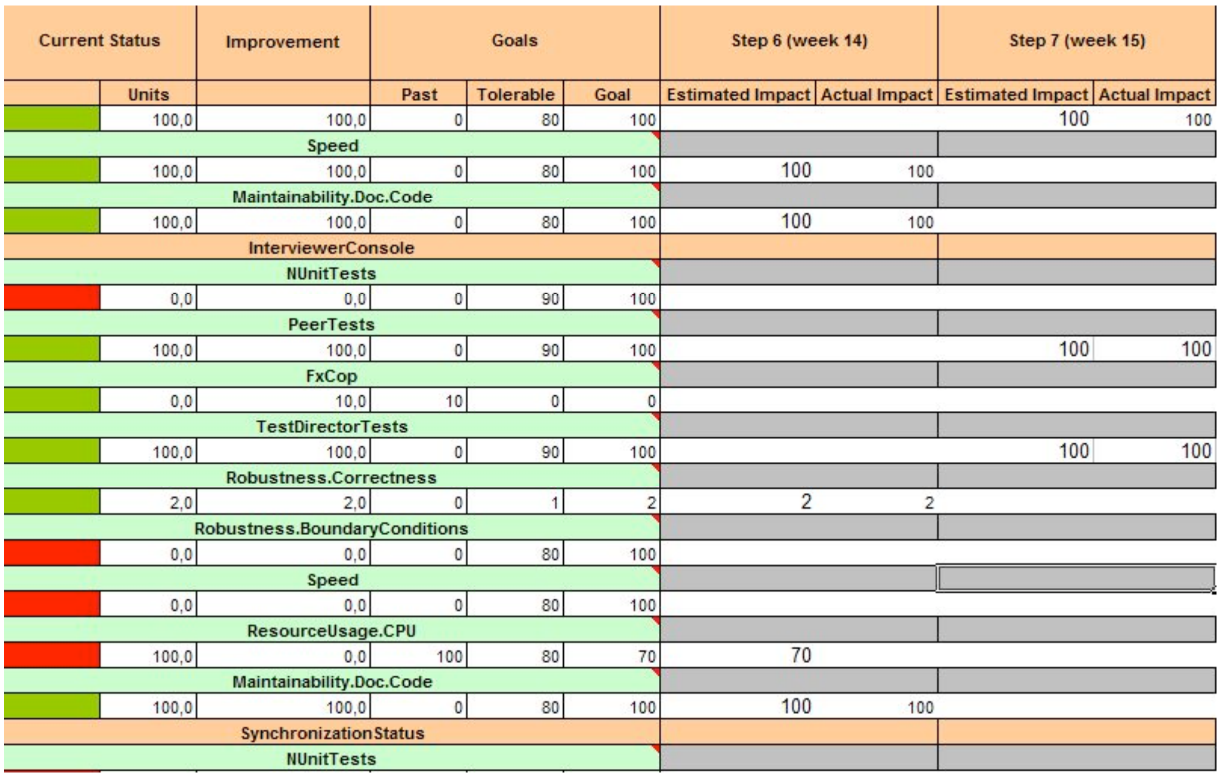


	…………………………Deliverables						
	Telephony	Modularity	Tools	User Experience	GUI & Graphics	Security	Enterprise
Business Objective							
Time to Market	10%	10%	15%	0%	0%	0%	5%
Product Range	0%	30%	5%	10%	5%	5%	0%
Platform Technology	10%	0%	0%	5%	0%	10%	5%
Units	15%	5%	5%	0%	0%	10%	10%
Operator Preference	10%	5%	5%	10%	10%	20%	10%
Commoditization	10%	-20%	15%	0%	0%	5%	5%
Duplication	10%	0%	0%	0%	0%	5%	5%
Competitiveness	15%	10%	10%	10%	20%	10%	10%
User Experience	0%	20%	0%	30%	10%	0%	0%
Downstream Cost Saving	5%	10%	0%	10%	0%	0%	5%
Other Country	5%	10%	0%	10%	5%	0%	0%
Total Contribution	90%	80%	55%	85%	50%	65%	55%
Cost (£M)	0.49	1.92	0.81	1.21	2.68	0.79	0.60
Contribution to Cost Ratio	**184**	42	68	70	19	82	92

Figure 4: an architectural level Impact Estimation table to try to understand the relationship between a set of high level objectives, and a set of architecture strategies ('deliverables'). 10% means an estimate that 10% of the level required would hopefully be delivered by means of that particular architectural element.

The architect can use the same basic method (Impact estimation) at a higher level of objectives and architecture, in order to understand the larger long term perspectives for the current architecture, to see the most promising architecture overall (for many objectives, and with regard to cost), and to 'sell' the architecture to management (what results are you going to get for the budget envisaged). This covers both the maintainability aspects and all other system performance aspects. We are a long way from having specialized software maintainability engineers, but it might be necessary one day.

Engineering Maintainability



Current Status		Improvement	Goals			Step 6 (week 14)		Step 7 (week 15)	
	Units		Past	Tolerable	Goal	Estimated Impact	Actual Impact	Estimated Impact	Actual Impact
🟩	100,0	100,0	0	80	100			100	100
Speed									
🟩	100,0	100,0	0	80	100	100	100		
Maintainability.Doc.Code									
🟩	100,0	100,0	0	80	100	100	100		
InterviewerConsole									
NUnitTests									
🟥	0,0	0,0	0	90	100				
PeerTests									
🟩	100,0	100,0	0	90	100			100	100
FxCop									
🟩	0,0	10,0	10	0	0				
TestDirectorTests									
🟩	100,0	100,0	0	90	100			100	100
Robustness.Correctness									
🟩	2,0	2,0	0	1	2	2	2		
Robustness.BoundaryConditions									
🟥	0,0	0,0	0	80	100				
Speed									
🟥	0,0	0,0	0	80	100				
ResourceUsage.CPU									
🟥	100,0	0,0	100	80	70	70			
Maintainability.Doc.Code									
🟩	100,0	100,0	0	80	100	100	100		
SynchronizationStatus									
NUnitTests									

Figure 5: One client of ours decided to use one week a month, every month to re-engineer the maintainability characteristics of their main software product (250 customers in 50 countries.)

The 'Green Week".

One client of ours, after trying to use one day a week to re-factor their software for about 2 years, decided to make use of a full week cycle, in a directly similar way that they developed the other quality characteristics of their product. The development team had a set of Scale-defined numeric targets (see "Goal" , figure 5) and constraints ("Tolerable"). They picked some of these to work towards, during one week each month. They did as well as they could, finding designs or processes that measurably gave them the levels of maintainability factors they required.

They thrived in an environment they called 'empowered creativity".

Summary

The many types of maintainability – ease of change – characteristics needed in large scale or critical software, can be architected and engineered using numeric measurement and sound engineering principles, instead of conventional small scale programming culture intuition. Real systems engineers will move towards this mode of 'real' software engineering.

We cannot continue to have the craft of programming culture, dominate our systems engineering practices – because software has become too critical a component of every major system. The real engineers have to take control. The programmers will not wake up without encouragement from real engineers.

References

Gilb, Tom, Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, ISBN 0750665076, **2005**, Publisher: Elsevier Butterworth-Heinemann. Sample chapters will be found at Gilb.com.

Chapter 5: Scales of Measure:

http://www.gilb.com/community/tiki-download_file.php?fileId=26

Chapter 10: Evolutionary Project Management:

http://www.gilb.com/community/tiki-download_file.php?fileId=77

Gilb.com: www.gilb.com. our website has a large number of free supporting papers , slides, book manuscripts, case studies and other artifacts which would help the reader go into more depth

INCOSE Systems Engineering Handbook v. 3

INCOSE-TP-2003-002-03, June 2006 , www.INCOSE.org

[Dart 93] Susan Dart , Alan M. Christie , Alan W Brown

A Case Study in Software Maintenance, Technical Report CMU/SEI-93-TR-8 ,

ESC-TR-93-185 , June 1993

Chris Inacio: Software Fault Tolerance, Carnegie Mellon University

18-849b Dependable Embedded Systems, Spring 1998

http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/

Google N-Version Software for more information on distinct software and N-version software.

Biography

BIOGRAPHY

Tom Gilb is an international consultant, teacher and author. His 9th book is '**Competitive Engineering**: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage' (August 2005 Publication, Elsevier) which is a definition of the planning language 'Planguage'.
He works with major multinationals such as Credit Suisse, Schlumberger, Bosch, Qualcomm, HP, IBM, Nokia, Ericsson, Motorola, US DOD, UK MOD, Symbian, Philips, Intel, Citigroup, United Health, and many smaller and lesser known others. See www.Gilb.com
.

Version 25 Oct 2007 start 1621 – end first draft 02:41 Oct 26th

Remember to include:

Crossman 27:1 reduction

External and internal stakeholders

Contracting and outsourcing